Miniature Vehicle Testbed for Intelligent Transportation Systems

Undergraduate Honors Research Thesis

By Jared D. Suter

Presented in Partial Fulfillment of the Requirements for Graduating with Honors Research Distinction in Electrical Engineering

Advisor: Dr. Keith Redmill

Department of Electrical and Computer Engineering

The Ohio State University

April 2016

Abstract

As it is often difficult and expensive to perform traffic pattern tests on full-scale vehicles, researchers require a smaller scale model to perform initial tests. Once researchers are confident in the results from the smaller scale tests, the tests can be performed on full-scale vehicles. Although small scale testbeds have been developed, this project investigated the development of a miniature scale testbed, the Miniature Vehicle Testbed for Intelligent Transportation Systems (MVT). The MVT is a tabletop test roadway system for initial tests of vehicle systems and traffic behavior. The testbed will provide a model for real life traffic situations and vehicle responses to those situations. Since no other testbed has been successfully implemented before on such a small scale, another goal of this project was to investigate whether such a testbed would provide a representative model for vehicle behavior, but this will not be possible until a full implementation and integration of the MVT is complete.

The MVT was built on a 4' x 8' platform such that it will provide an accurate model of real traffic patterns, while being an effective and simpler to operate replacement for the current larger testbed in Dreese Laboratories. The development of this testbed was approached in three separate primary phases:

1. Development of vehicle control software.
2. Development of imaging processing software to implement simulated GPS.
3. Construction of the physical testbed and testing.

The MVT, however, is incomplete in development. There are several issues with the implementation that still require a significant amount of time and attention to resolve. However, no evidence exists to suggest that the successful implementation of such a system is not possible. Further work on this miniature scale testbed is required, but the concept seems promising.

Acknowledgments

I would like to thank Dr. Keith Redmill for all of his guidance and support as I have worked on this project. I would like to thank Dr. Benjamin Coifman for participating in a thesis defense and for providing useful feedback on the project. I would also like to thank The Ohio State University College of Engineering for allowing me to use the university resources in Dreese Laboratories, and for providing the necessary supplies for my project. I would also like to express my gratitude for the support from my parents, David and Susan Suter, throughout my education at Ohio State particularly as I have worked on this project. I also would like to thank the ZenWheels company for providing me with an index of controls necessary to control their Bluetooth vehicles.

Table of Contents

List of Figures

List of Tables

Background and Motivation

Automotive research and development requires many different types of testing regarding a wide range of vehicle attributes and traffic conditions. One type of testing in the automotive field deals with traffic patterns. Researchers in this area are looking for things such as vehicle spacing, vehicle action and reaction timing, and collision avoidance in different traffic situations. For example, tests can assess how long it takes for a line of vehicles stopped at an intersection to clear the intersection, or the spacing between the vehicles as they begin to accelerate. To run these sorts of tests using full-scale vehicles requires a controlled environment. This would either require shutting down roads or having large on-campus facilities for such tests. Neither option is convenient or practical in a city such as Columbus, Ohio. In addition to this, acquiring full-scale vehicles and retrofitting them for testing is expensive, and the tests require larger crews of researchers to operate the vehicles and make proper observations.

The current answer to this problem is a smaller scale testbed located in Dreese Laboratories, shown in Figure 1. However, this testbed still occupies a very significant amount of space. It spans about 19.5 feet by 35 feet occupying 682.5 square feet. While it is much less expensive and easier to operate than full-scale tests, the smaller scale testbed still presents several issues. In particular, the layout of the testbed is severely limited by the dimensions of the lab in which it is located. For example, some of the features of the roadway have

Figure 1: The current small scale testbed in Dreese Laboratories

abnormal configurations. The number and variety of features are also limited, and it is difficult to implement changes in the model. In addition, the "vehicles" used on the testbed are actually Roomba vacuum robots shown in Figure 2, which are not ideal models for cars. Despite these drawbacks to the model, researchers have used this model to simulate traffic situations with reasonable accuracy, and have used this system for many years to gain insight on traffic patterns, thus validating the ability to gain useful information from small scale systems.

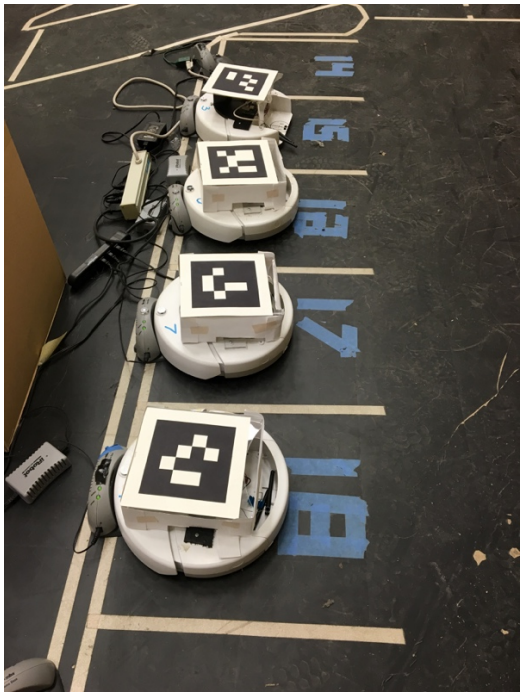


Figure 2: The Roomba vacuum robots used in the current testbed

Researchers desire a new model that will address many of the drawbacks to the current model while still providing a reasonably accurate model for real-life traffic. Such are the goals of the Miniature Vehicle Testbed for Intelligent Transportation Systems (MVT). In particular, the MVT will be implemented on a much smaller platform that will provide a more flexible layout and can be easily portable. The miniature scale testbed should also be more easily modified, and should provide a more ideal model with actual miniature vehicles.

Design Principles

The basic design of the MVT includes the same concepts as previously implemented systems, including the current small-scale testbed in Dreese Laboratories. Such systems are implemented using feedback-control systems, providing a system for input to vehicles based on desired position and previous motion. A block diagram for the feedback control implementation proposed for the Miniature Vehicle Testbed for Intelligent Transportation Systems is shown in Figure 3.

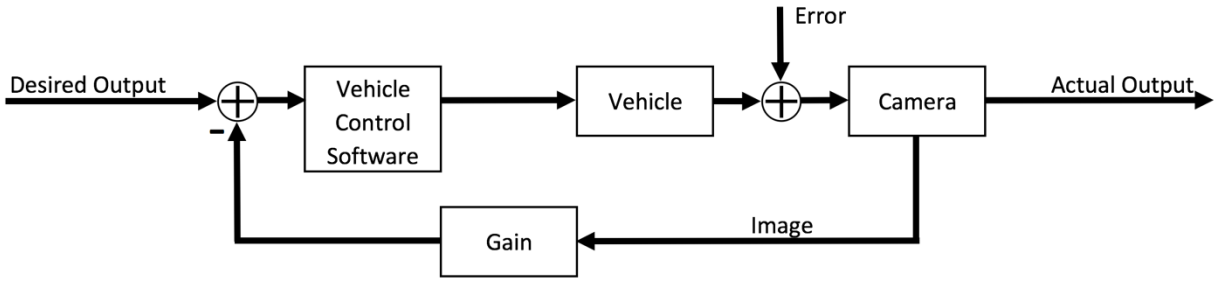


Figure 3: Feedback Control Block Diagram

The system is implemented using vehicle control software, written using the C programming language with support for up to 10 ZenWheels micro cars. The ZenWheels micro cars are Bluetooth controlled, and were chosen because of their advertised precision in control. One such vehicle is shown in Figure 4. Initial tests with these vehicles in experimentation suggest that control of these vehicles is indeed very precise. These vehicles also include several additional features that allow for possible additions to the model, such as a magnet detector on the bottom of the vehicle to detect if the vehicle drives over a metal strip, or several sets of lights on the vehicle that function as turn signals, headlights, and brake lights. The system also includes a high-resolution camera used for simulated GPS. When a desired output, in this case a desired location for the vehicle, is input to the system, the vehicle control software calculates the actuator parameters for the vehicle, specifically the "throttle" (forward or reverse motion),

"steering" (to initiate the action of turning), and "trim" (correction for unwanted steering in the vehicle) if necessary. After a certain amount of time has passed, the vehicle has traversed a path which will most likely include some error. This error could be due to error in actuator control in the vehicle with the servo motors, or it could be due to forces in the testbed unaccounted for in the model such as slipping wheels or bumps on the road. To correct for this error, the computer fetches an image from the high-resolution camera mounted above the system. The vehicle control software runs an image processing algorithm on the fetched image to locate unique tags attached to the vehicles on the testbed and attached to other locations of interest on the testbed. From the image processing algorithm, the vehicle control software determines the location and orientation of the vehicle in question. Using this information, the vehicle control software applies a gain to determine where the vehicle is moving, and uses this feedback to re-calculate the actuator parameters from the vehicle. Conceptually speaking, the current location of the vehicle is subtracted from the desired location in each feedback loop. This feedback loop continues until the vehicle's current location is equal to the desired location. These components of the system are discussed in further detail in following sections.

Figure 4: A ZenWheels micro car used on the testbed

Physical Testbed Construction

The MVT was designed to be implemented on a 4-foot by 8-foot surface. The MVT reflects several design considerations, including portability, structural integrity, and flexibility in road layout, as well as allowing imaging using conventional cameras and image processing software. The MVT is built on a 4-foot by 8-foot sheet of ¼-inch finishing plywood. This yielded a smooth, durable surface that was easily paintable. To supply rigidity, the platform is framed using 2-inch by 2-inch wood studs. Since a 4-foot by 8-foot structure could be difficult to handle and move, the platform consists of two separable 4-foot by 4-foot pieces. The two halves are aligned using holes in the 2 x 2 studs on the joining faces, with ½-inch dowels. Once aligned, the platform pieces are held together using two metal hasps on the side.

To prevent the vehicles from falling off the side of the testbed platform, ½-inch quarter-round trim is mounted around the edge of the platform. This also provides an aesthetic appeal for the platform. The platform is painted black on the underside, and green on the top side, using flat finish paint to reduce glare on the surface that might affect the image processing algorithm. Initial testing of the image processing algorithm suggested that glare on the surface could create a lot of noise in the image, which could affect how well the image processing algorithm operates. The green on the top side was designed to mimic the color of grass, in an attempt to create a sense of realism for the model. A picture of one half of the structure is shown in Figure 5.

Figure 5: One half of the testbed platform structure

The roadway design was developed in consultation with Dr. Redmill. The design maintains a reasonable scale for the road compared to the size of the vehicle. The ZenWheels micro cars are approximately 2 inches long, 1 inch wide, and 1.25 inches tall. These dimensions do not translate well to average vehicles, but they do approximate the dimensions of a compact car. Using the width as a benchmark, the micro cars are approximately 1:63 scale of a compact car. Typically, road lanes are more than 10 or more feet wide, so to make design of the road easier the lanes are 2 inches wide, which corresponds to slightly more than 10 feet.

The roadway system was designed to include as many features both from the existing small-scale testbed and from real-life roads. Some examples of features included are curves, multiple lane highways, long spans of straight roads, a 3-lane road with a center turn lane, several 3-way and 4-way intersections, a 5-way intersection, a traffic circle, a parking lot, continuous exit ramps, several areas with merging lanes, a bridge, and an underpass. These features are all easily found on roads in the area surrounding Columbus. These features were included so that the testbed

could be an accurate and versatile model of roadway systems. Appendix A features the testbed roadway design.

The roads were cut from heavy duty dark-gray card stock. Once the roads were cut, lane markings and other features were painted using Sharpie oil-based paint pens. Similar to actual roads, the edges of the roads were painted white, and the dividing lines were painted yellow. A ZenWheels micro car is shown on a piece of the road in Figure 6.

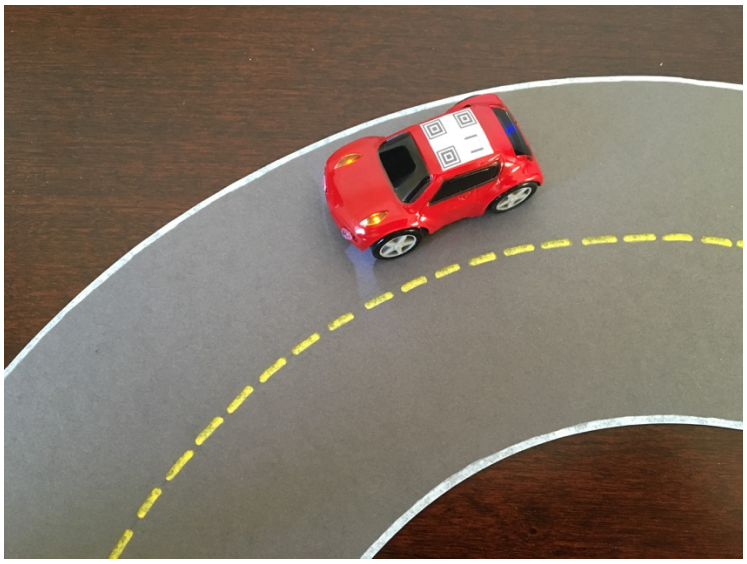


Figure 6: A micro car on a section of road for the testbed

The bridge for the road was constructed to span the width of the testbed. It was built with 20 inches of span on the top, 6 inches wide to accommodate two lanes and with two ramps on either side. Since one of the goals for the testbed is portability, the bridge was carefully designed so that it could be disassembled. Each of the ramps can be detached from the bridge, and the center support for the bridge can be detached as well. This is important because the bridge was constructed using soft modeling woods such as balsa wood, which can easily be damaged. If the bridge is transported in several pieces, it is less likely to be damaged. Currently the bridge has

not been painted, but for the final testbed it will be spray-painted black. A picture of the bridge is shown in Figure 7.

Figure 7: The bridge for the testbed

To mount one or more cameras on the testbed, it was necessary to construct some sort of structure above the testbed. Theoretically, the high-resolution camera has a viewing angle of about 60 degrees, 30 degrees on each side. Using some simple trigonometry:

$$Camera\ Height = \frac{2'}{tan(30°)} = 3.46'$$

where 2 feet corresponds to half the width of the testbed, and 30 degrees is the viewing angle that will cover this distance. This suggests that the height of the camera should be about 3 and ½ feet. To allow for significant tolerance, the camera frame was constructed about 5 feet above the testbed.

The camera mounting frame had some simple requirements. First, to achieve portability the camera mounting frame had to be easily disassembled. Second, it had to be lightweight for easy transportation. Third, it had to be sturdy enough so that the camera wouldn't experience significant swaying from a flimsy structure. Finally, the frame should be capable of modification to accommodate different cameras or different camera layouts. To achieve these goals, the camera mounting frame was constructed using PVC plumbing pipe. To attain the maximal amount of sturdiness while still being lightweight, ¾ inch PVC pipes were used. The pipes were mounted onto the side of the testbed platform using 4 T-junction PVC fittings. These fittings were screwed into the side of the testbed so that the upright pipe for the frame would be held vertically. This provided a high amount of sturdiness in the structure. To accommodate two cameras for the testbed with one camera centered over each side of the testbed, these fittings were mounted directly in the middle of each side of each half of the testbed. The 5 foot upright pipes were then inserted into the fitting mounts and supported an "H" frame above the testbed. The 90 degree fittings that are attached to the H frame were all cemented to keep them from moving. The rest of the joints simply use a pressure fit, so they can be easily disassembled for transport. The testbed with the camera mount frame is shown in Figure 8. The high resolution cameras are then mounted onto the frame using camera mount clamps, shown in Figure 9.

Figure 8: The testbed platform with the camera mounting frame

Figure 9: The camera mount clamp and camera

Image Processing Algorithm

Matlab was used for initial development of the image processing algorithm. Matlab provides several useful tools such as the Image Processing Toolbox and the Matlab Coder, as well as many useful debugging features that come with a high-level Integrated Development Environment (IDE). Developing an effective image processing algorithm proved by far to be the most difficult part of the testbed.

First a suitable tag had to be developed to use as a unique identifier for the vehicles and for points of interest on the testbed platform. Several versions of the tag were developed, experimenting with different features. The tag had to be such that it would be possible to easily determine both the location and the orientation of the tag. Initial tags were tested having different shapes such as triangles which were supposed to show the orientation of the vehicle, or alphanumeric characters that would explicitly show the identification for the tag. Two examples of initial tags considered are shown in Figure 10 and Figure 11. From the initial designs, it became clear that slanted lines in the design from triangles cause extra pixilation in the original image, and essentially adds undesired noise in the image before it is even processed. Ideally, the suitable tag would feature straight vertical and horizontal lines, with easily identifiable and unique tags based on a pattern. The final tag being used more closely resembles that of a QR code, as shown in Figure 10. It features 3 corner tags consisting of nested squares of alternating black and white color. The identifier of the tag consists of a 5-bit binary code. This code is created by upright bars, and a line underneath to show an observer what the code should be. How the image processing algorithm determines the identifier will be explained later. Tags were

printed for the vehicles at 10% scale of the original image created on the computer, and tags used on the board were printed at 30%. A tag on top of a micro car can be seen in Figure 6.

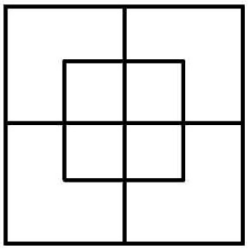


Figure 10: An initial tag design featuring squares

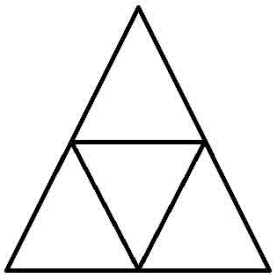


Figure 11: An initial tag design featuring triangles

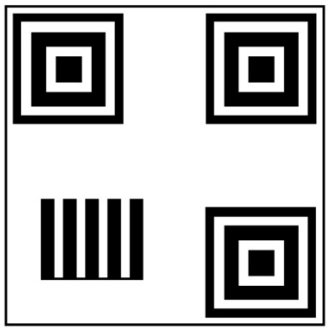


Figure 12: The testbed tag template

Next, a suitable algorithm for detecting the created tag in an image was determined. Harris-Stephens Corner Detection was used to detect the several corners in the nested square tags in the corners. Harris-Stephens Corner Detection operates using a small "window" through which the intensity of the image is analyzed. This window is used to detect corners by calculating the change in intensity as seen through this window as the window is moved in different directions. If the intensity of the image changes as the window is moved in any direction, then the point is

determined to be a corner.[1] Ideally this creates an 'X' pattern of detected corners. Since the Harris-Stephens Corner Detection will create a list of corners including much more than just the desired corner patterns, the best approach is to continue processing smaller sets of corners rather than the entire list of corners. These smaller sets are created using a technique known as "clustering." Clustering is the action of creating smaller sets of points based on proximity to each other. Once the clusters are created, an algorithm is applied to detect lines of points. Once the lines are detected, they are checked to see if they match the 'X' pattern. Once three such symbols are found within the correct proximity, they can be used to determine the orientation and location of the tag. They are also used to determine the location of the identification tag, which is then read.

Appendix B includes the image processing algorithm as developed in Matlab. The image processing was developed in three parts, findTags.m, TestbedTag.m, and harris.m. Original attempts to develop this code used custom implementations for clustering and for line detection. The original attempt at clustering created an unlimited amount of clusters by adding points to the clusters within a minimum distance to the cluster centroid, which is the average location of the points already contained in the cluster. However, this algorithm for clustering proved to fail when corner points were too close to the desired tag, so that the clusters would be moved significantly from their desired locations and sometime the nested square tags would even be split into two smaller clusters. This was solved by using a native Matlab function, kmeans(), instead, which implements an algorithm known as k-means clustering. The K-means clustering algorithm begins by using K points in the space as initial centroids, which are cluster centers.

[1] Reference to "Berg, Alex." (See Bibliography)

Each object is then assigned to be in a cluster with the closest centroid. Once all the objects are assigned the centroid positions are recalculated using the average of the objects in the cluster. The process of reassigning objects to clusters is repeated until the centroids do not move.[2] K-means clustering created the clusters much more reliably.

The original attempt at line detection used an algorithm to check for lines that passed through the center of the cluster. This was effective at finding the lines when the cluster contained only the nested square tag without a significant number of extra corner points. However, this proved to be effective only for an idealized image, and was ineffective for most situations. The solution to this problem was to implement an algorithm known as Random Sample Consensus (RANSAC). RANSAC is a resampling technique that begins by randomly selecting the minimum number of points to determine the model parameters, in this case 2 points to create a line. The parameters of the model are then calculated, in this case the equation of the line. Next the number of points that support the model by a given tolerance is found. If the number of inlier points is above a defined threshold, the selected points are used as the correct model. Otherwise, the process is repeated using new points.[3] This algorithm proved to be much more effective for any input image.

The Matlab code findTags.m is the main code for the image processing algorithm. This code takes an input image, applies the Harris-Stephens Corner Detection to find all of the corner features in the image, uses k-means clustering to form clusters from the detected corners, and

[2] Reference to "Clustering – K-means." (See Bibliography)
[3] Reference to "Derpanis, Konstantinos G." (See Bibliography)

applies RANSAC to find the strongest line in each cluster. Once it finds the strongest line, it searches for lines that are perpendicular to that line. If it finds a suitable line that is perpendicular to the first line, then the algorithm treats this as having found a nested square symbol in the image. The algorithm finds all of the clusters that contain nested square symbols, and then groups those symbols based on proximity. If the algorithm finds three of these nested square symbols within an appropriate proximity, the algorithm treats this as having found a tag. The algorithm then determines which two symbols form the "hypotenuse" of the tag, which is the imaginary line that crosses the middle of the tag. The center point of this line is the center of the tag, which is used as the location of the tag. Simple trigonometric calculations then determine the orientation of the image given the slope of the "hypotenuse" and the third symbol. Once the orientation and location are known, the image processing algorithm can determine the location of the bars used in the identifier tag. This is done by using a ratio of the calculated distance between the nested square symbols in the image to the actual distance in the original printed image. Both of these values are calculated as a number of pixels. Using this ratio and some simple trigonometry, test points to check which bars are present are generated. Since the image is in gray scale, if the bar is present then the value of the image will be a low value, and if the bar is not present then the value of the image will be a high value. Checking these values can determine the binary code corresponding to the bars present. This binary code is then translated into a decimal identification number. FindTags.m returns the ID, location, and orientation of any observed tags in the image.

The files testbedTag.m and harris.m were created as helper functions for findTags.m. Harris.m includes a Harris-Stephens Corner Detection algorithm, adapted from code found online and

optimized for use with the testbed. In particular, the original algorithm returned a large number of false corners near the edge of the image, so code was added to the algorithm to remove all corners within a certain distance to the edges of the image. TestbedTag.m is a classdef file that created an object called testbedTag. This file included a definition for the object, which included an ID value, location coordinates, and an orientation value. TestbedTag.m also included a constructor method, and a method used to create an array of defined size of testbedTags. FindTags.m uses the testbedTag as the return type.

The code includes several parameters that need to be adjusted and optimized for the input image. While this would be an impractical solution for a dynamic system, the testbed is assumed to be a static system in that the operation parameters of the system should never change. Specifically, the distance ratio on the board from pixels to inches should never change because the cameras are statically mounted above the testbed. Since the tags are also of static size, there should be no change in parameters once the testbed is setup. Changing the parameters for every input image made testing more challenging, but is entirely reasonable for the final testbed implementation. If the testbed is ever redesigned, it should also be easy to adjust these parameters for the new system. Some of the adjusted parameters are the number of clusters to be created by the k-means algorithm, the distance threshold for a point to be considered part of the line in the RANSAC algorithm, the number of support points necessary for RANSAC to consider a line valid, and ranges of reasonable lengths for the nested square symbols within the tags.

The results from a test image in Matlab are shown in Appendix D.

Vehicle Control Software

The MVT uses a Linux computer in the Dreese Laboratories Intelligent Transportation lab. This computer is equipped with a Bluetooth dongle to allow it to communicate with the ZenWheels micro cars. The computer is also the host computer for the vehicle control software, which is responsible for initializing the camera, reading in an input program, connecting the micro cars, fetching images from the camera, calling the image processing algorithm routine, and calculating the actuator output for the micro cars based on desired location and current location. The software was developed using the C programming language in the gedit text editor.

The first part of the vehicle control software is the initialization of the camera. The camera used is an ELP 5.0 Megapixel USB camera. Source code that was provided with the camera by Dr. Redmill to initialize the camera in the vehicle control software. This part establishes a link from the camera and provides a method for fetching images. Once this is complete, a simple algorithm is used to fetch an image from the camera input stream.

Next, the vehicle control software prompts the user for the name of a test input file. Once the user provides the name of a valid text file containing the test routine, the software begins to parse the file into an array of command codes. Another program, test_program_generator.c, is used to automatically generate a test file. This program prompts the user to input all the desired commands, verifies the commands, and prints them in the correct format in an output text file. This text file of a specific format that encodes the input commands into something that is readable by the vehicle control software. The first line of each test file is an integer expressing the number of cars that are used in the test, which can range from 0 to 10, and the second line of

each test file is a positive integer expressing the number of commands to be used in the test. Both of these values are stored as variables in the vehicle control software. The next lines define the starting positions of each vehicle, and the starting states of any intersections that are to be initialized. After all the initialization is complete, the test file marks the beginning of the commands with a '0'. Following this, all the test commands are printed in the file using the coding structure:

'vehicle' 'command' <'operand 1'> <'operand 2'>

such as the command for vehicle 1 to drive to coordinates 16, 5:

1 d 16.00 05.00

where the coordinates are expressed in inches. The MVT uses inches rather than a metric scale because the scale of the vehicles compared to real life roads is more easily expressed in inches. Not all commands include operands. Table 1 shows all the possible commands and their structures.

Code	Command	Operand 1	Operand 2	Action
d	Drive	Destination X	Destination Y	Drive directly to the destination
c	Curve	Destination X	Destination Y	Drive around a curve to the destination
t	Traffic Circle	Exit X	Exit Y	Drive around the traffic circle to the exit
l	Left	Destination X	Destination Y	Turn left at an intersection to the destination
r	Right	Destination X	Destination Y	Turn right at an intersection to the destination
s	Straight	Destination X	Destination Y	Navigate straight at an intersection
w	Wait	Time (seconds)	0	Wait for a given number of seconds - idle
p	Park	0	0	Park the car - end of program for vehicle

Table 1: All possible commands and their structures

Using the input test file, the vehicle control software parses the commands into a matrix of commands with numerical values. During this process the vehicle control software also initializes a program counter array, which tracks the current command to be executed for each

vehicle. Using this system, the software ensures that all the commands are executed in order. A commented sample test file is shown in Appendix C.

After the input test file is parsed, the software then would import any necessary lookup tables. This portion of the code has not been implemented yet, but it would function in a similar way to parsing the test file. One lookup table of particular interest would define the states for the intersections on the testbed, and would include information about their location and connected road segments.

The next step in the code initializes the connection to all of the used vehicles for the test. This is done using a switch case statement for the number of cars, so the cars are connected according to the number of cars needed in reverse order. Once each vehicle is connected, the software verifies the socket that the vehicle is connected to, and turns the "running lights" on on each vehicle. This part of the code initializes an array of Bluetooth socket numbers that are used to control the vehicles. Commands are sent to the vehicle using the following format:

```
char c[10];
c[0] = CHANNEL; c[1] = VALUE; send(vehicles[CAR],c,2,0);
```

where c[10] is the serial command to be sent to the vehicle, CHANNEL is the serial channel corresponding to the desired action, VALUE is the code for the desired action to occur, and vehicles[CAR] corresponds to the Bluetooth socket connected to the desired vehicle. Table 2 lists all of the CHANNEL and VALUE codes and their corresponding actions. Some of the information from Table 2 was adapted from the information provided by ZenWheels.

Channel	Parameter	Value	Action
0x81	Steering	0x00 - 0x3F	Steer right (0x3F sharpest)
		0x40 - 0x7F	Steer left (0x40 sharpest)
		0x00	Steer straight
0x82	Throttle	0x00	Stop
		0x01 - 0x03	Forward, but no motion (low power)
		0x04 - 0x3F	Forward (0x3F max speed)
		0x40 - 0x7C	Reverse (0x40 max speed)
		0x7D - 0x7F	Reverse, but no motion (low power)
0x83	Left Turn Signal	0x00	Off
		0x01	Front On
		0x02	Front Dim
		0x04	Rear On
		0x08	Rear Dim
0x84	Right Turn Signal	0x00	Off
		0x01	Front On
		0x02	Front Dim
		0x04	Rear On
		0x08	Rear Dim
0x85	Headlights	0x00	Off
		0x01	On
		0x02	Bright
0x86	Horn	0x00	Off
		0x01	On
0x87	Light Effects	0x00	Off
		0x01	Low Glow
		0x02	Phasing glow
		0x03	Police 1 - with siren
		0x04	Police 2 - with siren
		0x05	Police lights - no siren
0x88 - 0x8C	Unused	-	-
0x8D	Trim	0x00 - 0x3F	Trim right (0x3F sharpest)
		0x40 - 0x7F	Trim left (0x40 sharpest)
		0x00	Trim straight
0x8E	Unused	-	-

Table 2: ZenWheels micro car command list

After the vehicles are connected, the vehicle control software calls the image processing algorithm to determine the actual initial locations of all the vehicles. This information is then used to initialize matrices that are used to track the current location information and current state information for the vehicles. These matrices are carLocations, and carStates. Both of these matrices have 3 columns, and they both have a row for each vehicle. The format of a row of each matrix is:

carLocations[car][:] = [current position X] [current position Y] [current orientation]

carStates[car][:] = [current throttle value] [current steering value] [current trim value]

The vehicle control software includes a do-while loop structure for this section to make sure that all the vehicles are found, and the code repeats until all vehicles are initialized.

The last step for setup of the test is to move all of the vehicles to their defined starting positions. For each vehicle that is being used, the setup routine moves the vehicle toward the desired starting position until it is within a defined acceptable distance from the starting point. This is done by using a do-while loop structure to execute the feedback control structure until the vehicle has "arrived." Inside this loop structure, the image processing algorithm is called, and the returned information is used to set variables for the current X and Y locations of the vehicle. Using this, calculates the distance of the vehicle from its desired destination using the distance formula:

$$\sqrt{(desired\ X - current\ X)^2 + (desired\ Y - current\ Y)^2}$$

and calculates the necessary change in orientation using the formula:

$$\frac{180}{\pi}\tan^{-1}\left(\frac{desired\ Y - current\ Y}{desired\ X - current\ X}\right) - current\ orientation$$

and the output steering value is calculated by increasing the current output steering value by the necessary change in orientation multiplied by a constant and expressed in hexadecimal notation. The throttle is set to be a constant low speed for this portion of the code.

Finally, the main test routine is ready to execute. As it stands, the code only support operation by one vehicle, but a planned solution using multithreading is discussed in more detail in the future work section below. The test operation code is fairly simple. The basic operation is performed by iterating through the commands for the vehicle being tested. Command execution

is tracked using the aforementioned array of program counters. For each command, the software executes a switch-case statement on the action operand. Depending on the action, the software executes a slightly different algorithm. If the action is “drive,” “curve,” “left,” “right,” or “straight,” the majority of the algorithm is the same as the algorithm for moving the vehicles to their starting locations. For “drive,” the vehicles travel at a default speed, unless they are within a certain threshold distance from the destination then they travel at a slow speed. These algorithms also include code to adjust trim in addition to steering. This is done by adjusting trim rather than steering if the calculated change in direction is less than a certain threshold. The main difference for the “curve,” “left,” and “right” algorithms is that the speed is set to the slow speed. “Straight” operates exactly like “drive.” “Wait” operates by translating the input time in seconds to micro seconds, and using the command usleep to suspend operation for the given amount of time expressed in microseconds. All commands except for “park” increment the program counter for the particular vehicle. For “park,” the code sends messages to the vehicle to stop and reset steering, and turns on the blue underglow lights to visually verify that the vehicle is parked. “Park” also sets the program counter for the vehicle to 0, which indicates that the vehicle execution has finished, and the program is complete. The final part of the vehicle control software closes the link with the vehicles, closes the camera feed, and frees any dynamically allocated memory used in the program.

Implementation Issues

Development of the MVT is not complete. Unfortunately, there are several significant issues with the testbed in its current state that will require much more time to resolve. This section will highlight some of the more prominent issues that have been discovered with the current implementation of the testbed, and suggested solutions and their expected outcomes. Thus, while unfinished, there are several aspects of the current testbed that suggest that it is possible to successfully develop and implement a miniature scale testbed.

The first major issue pertains to the camera mounted above the testbed. The required height for the camera to be mounted was calculated assuming that the viewing angle of the camera is 60 degrees. As discussed above, the calculated required height was equal to be about 3.46 feet, and the camera was mounted at 5 feet above the testbed to allow for an error margin. However, even mounted at 5 feet the camera still does not cover the width of the testbed as seen by the measuring tape in Figure 11 below, which is spanning the width of the platform from 0 to 48 inches. The approximate height necessary for the camera is 5 feet and 6 inches, which means the actual viewing angle of the camera must be:

$$2 * \tan^{-1}\left(\frac{2'}{5.5'}\right) \cong 40°$$

approximately equal to 40 degrees. Furthermore, from a distance of 5.5 feet the camera may not have a high enough resolution to capture sufficient detail to identify the symbol attached to the top of the vehicle. In fact, the resolution may not be high enough to view the larger tag on the testbed. The images of these objects taken from the camera mounted at 5 feet above the testbed are shown in Figure 12 and Figure 13. These tags are not sufficiently clear in these images to be

effectively detected in the image processing algorithm. These images also suggest that exposure may be an issue causing the tags to be unreadable. This can be observed from the fact that the tag has become completely white in Figure 12. This issue is likely controlled as a parameter of the camera, but some further debugging and testing would be required to determine the best solution. To address the issues caused by distance from the tags, increasing the size of the tags would not be a practical solution because the size of the tags mounted on top of the vehicles would become unrealistically large and would impede the motion of the vehicles or would cover the vehicles completely. The size of the tags on the testbed could not be significantly increased because of a lack of unused space on the testbed surface.

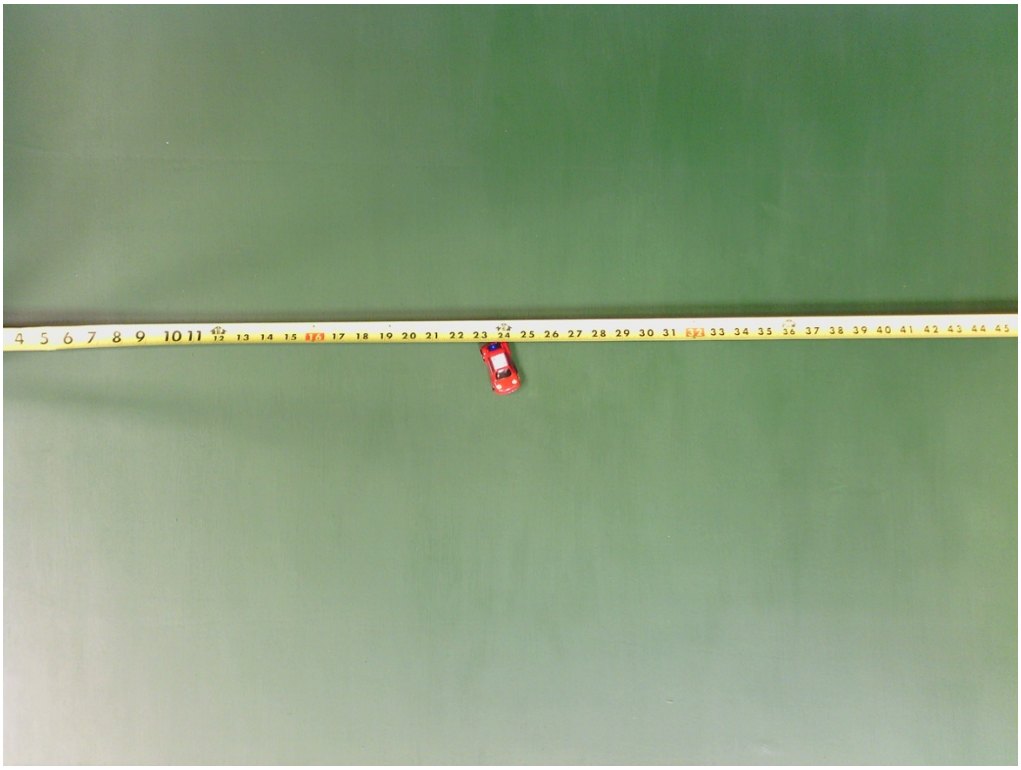


Figure 13: A snapshot of the testbed using the mounted camera

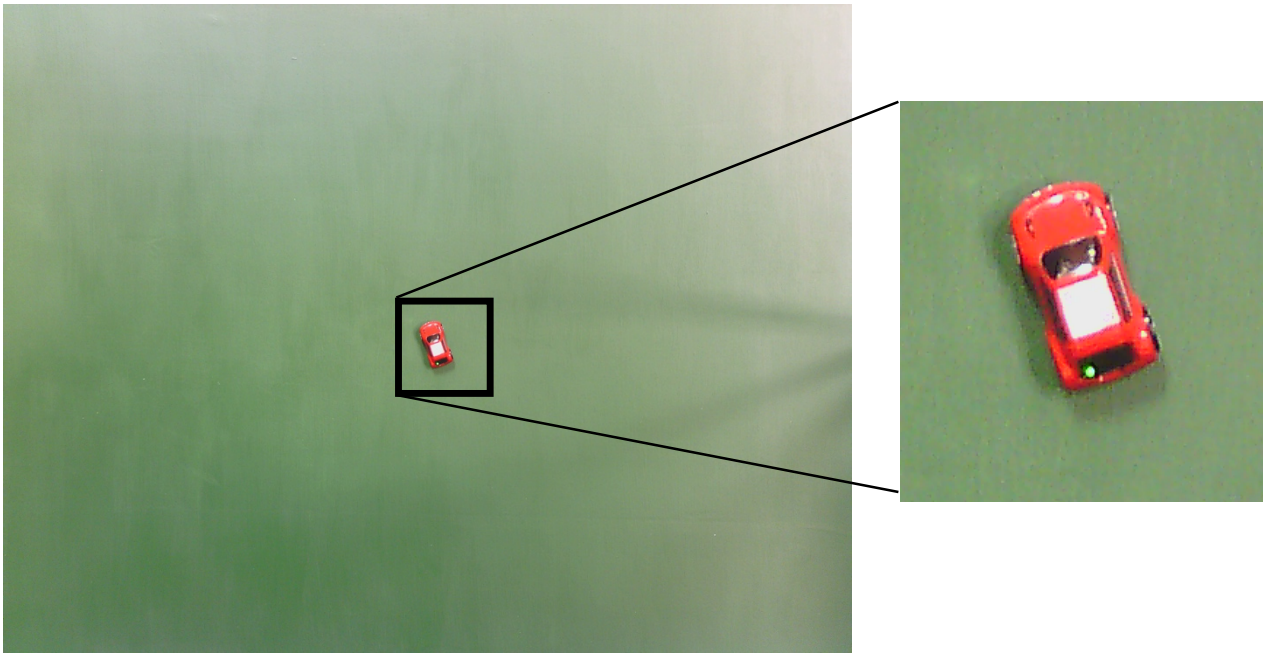


Figure 14: A snapshot of a vehicle from the camera mounted above the testbed

Figure 15: A snapshot of the larger testbed tags from the mounted camera

This particular problem has two possible solutions. First, a higher resolution camera with a wider field of view could be obtained. This should theoretically cover the span of the testbed with a much lower mounting height, and the resolution should be high enough to have a reliably

clear image of the tags. Alternatively, a set of the same or similar cameras could be used, but a higher number of them at lower heights. For example, if 6 cameras were used, each camera would be responsible for covering 1/6th of the board, which corresponds to a 2-foot by 2-and-2/3rd-foot section. Given the field of view of 40 degrees for the camera, the new mounting height would be:

$$Camera\ Height = \frac{1'}{tan(20°)} \cong 2.75'$$

approximately 2.75 feet, which is approximately half the original height. The set of cameras collectively should span the entire board, and much more detail would be captured by the cameras in the available resolution. If this solution were employed, it would be necessary to modify the camera mounting frame so that the frame had bars above the centers of each of the 6 subsections of the board. As the frame is constructed with PVC pipe, it is readily modified. It would also be necessary to capture images from all 6 cameras, and fuse them together in the software before running the image processing algorithm on the compiled image. Since this issue has at least two likely viable solutions, this issue should not be an obstacle to successful development of the MVT.

Another camera-related issue concerns the image fetching algorithm. As it stands, the image is read from the camera feed, and saved to a file labeled "testbed.ppm." During system testing, after running the algorithm to fetch an image from the camera feed the saved image would remain unchanged for a few iterations. Another similar issue is the unpredictability of the camera's lens focus. It appears that the focus of the lens as adjusted during camera operation, but it is unclear exactly how the focus is controlled. Since the refocusing is uncontrolled, it is unpredictable in current operation. This unpredictable refocusing could cause fetched images to

be blurry, and potentially unreadable by the image processing algorithm. Since images would have to be fetched and processed in real-time in the final testbed implementation, any such issue that would cause delay in the software execution is unacceptable. The solutions to these problems are not clear at this time, but will require some debugging and code tracing to determine the source of the issues. Since the camera implementation is the same as used in many other systems including the existing testbed, viable solutions to these issues exist and should not be a permanent obstacle to the successful development of the MVT.

Another significant issue pertains to the image processing algorithm software. More specifically, the k-means clustering algorithm introduces an undesirable randomness in the cluster generation. The same image with the same parameters could result in successful detection of the tag in one test, and then result in a failure in the next test. This is because the initial points for the cluster creation are selected from the total set at random. Another aspect of the k-means clustering algorithm that presents some issues is the fact that it requires the number of desired clusters to be entered as a parameter. The issue with this is that having too many clusters could cause the nested-square symbols to be split into separate clusters, while having too few clusters could result in more than one nested-square symbol being included in a distinct cluster. Matlab output where one of the nested-square symbols is split into two separate clusters is shown in Figure ___. In this example, splitting the nested-square symbol to be split into two clusters kept the software from identifying the symbol. This creates an unreliability factor, which would only become more complicated as more tags are added to the testbed. Potentially this issue could become worse as tags move closer or further away from each other and the clusters shift with them. A solution to this issue may be to incorporate a size and proximity limitations on the clusters

created in the clustering algorithm, similar to the initially attempted algorithm. Another solution might be to rewrite the nested-square tag detection algorithm so that it considers the possibility that multiple symbols may be in the same cluster, and possible combine adjacent clusters to see if any symbols were split. Since this issue has at least two likely viable solutions, this issue should not be an obstacle to successful development of the MVT.

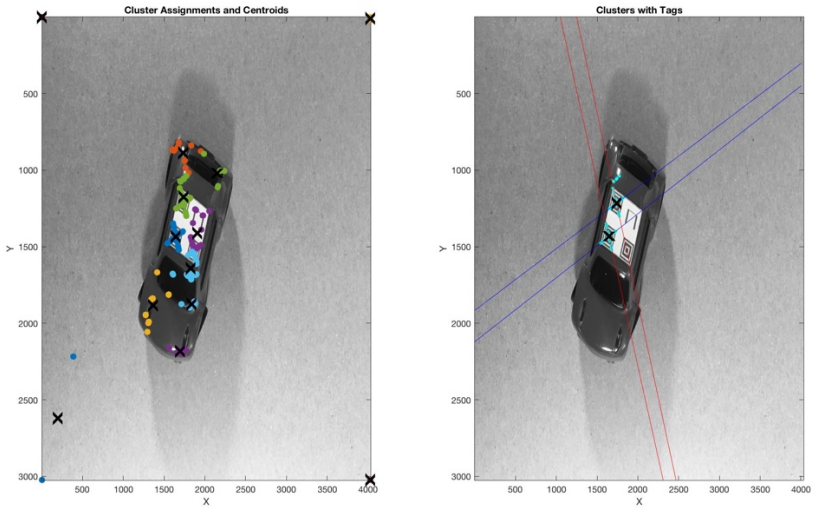


Figure 16: RANSAC error output

Another issue with the image processing software pertains to the detection of the 'X' pattern in the nested-square symbols. The determination of the 'X' is dependent on finding the strongest line according to a RANSAC algorithm, and then finding the strongest perpendicular line using the RANSAC algorithm again. The perpendicular line is found by taking the negative inverse of the slope of the original line, and the calculated perpendicular slope is used to model the line that is being looked for. However, there have been situations where some extra points are included in the cluster that are close to the nested-square symbol, and the first line detected is not quite the correct line, and the second line is not detected as an appropriate match. This issue can likely be solved by either creating a margin of error for the calculated slope of the second line, or by

creating a loop that would recheck the relationship between the first and second lines if a second line is not detected. Another potential solution might be to detect all suitable lines in the cluster and try to match the lines that are approximately perpendicular. Since this issue has at least two likely viable solutions, this issue should not be an obstacle to successful development of the MVT.

Another issue pertains to the translation of the Matlab image processing algorithm into C source code. To accomplish this, Matlab includes a tool called "Matlab Coder" which automatically generates C source code from Matlab source code. It took significant work with editing and reworking code to get the Matlab coder to generate code. This was because there are features of the code in Matlab that are incompatible for translation to C. For example, the function "corners" which was originally used as a native Matlab function to implement Harris-Stephens Corner Detection is not supported for code generation. This function had to be replaced with a customized Harris-Stephens Corner Detection algorithm, harris.m (included in Appendix B). Once the Matlab coder finally generated code, a significant amount of time was put into compiling the code. This was because the automatic code generation included references to Matlab libraries, and some issues with access to these libraries had to be solved. Eventually the code was compiled, but does not function properly. To debug these issues, the source code needs to be analyzed, and this can prove to be difficult as the automatically generated code is very cryptic.

Unfortunately, several of these issues, as well as many issues that were already solved, have caused significant delays in the progress of the project such that the project has not yet been

finished. Some pieces of the project have not been rigorously tested yet and there may still be more bugs that have not been uncovered. However, the underlying code for several pieces feature functioning algorithms, with some relatively minor bugs that need to be fixed. In particular, many instances of successful tag detection for several input images have been recorded using the image processing algorithm. The solution of these issues remains as part of the future work for this project.

Future Work

Obviously, the work for the immediate future on the MVT will be to finish development of the testbed. The first priority for the future work will be to remedy the issues documented in the previous section. Once these issues are fixed, those respective portions of the MVT system should be about complete. That said, there still remain several other smaller issues to be dealt with and several other features that remain to be implemented. Even once the initial development of the testbed is complete, there are many ways that the system can be expanded in the future.

The following is a list of the known tasks for initial development work:

- The physical construction of the testbed is still not complete. Many card stock road pieces still need to be drawn, cut, and painted. The camera mounting frame will most likely need to be rebuilt as described in the previous section. The camera mounting frame and the bridge still need to be painted black, and some touchup paint on the testbed platform still needs to be done.

- A lookup table for the intersection state definition still needs to be created. This table will list all of the intersection on the testbed, as defined by a map, using capital alphabetical characters (e.g. A, B, C, etc.). For each intersection defined, there will be a number of states for that intersection using positive integer values. For each state, there will be a set of coordinates that correspond to the location from which vehicles are allowed to enter, and a set of actions that they will be allowed to take. Each state could have multiple lines corresponding to multiple directions being active at a time. Each

state will also have a line that defines how long the state is active in a cycle. An algorithm for importing this lookup table needs to be developed in the vehicle control software, which should be similar to the algorithm for parsing the test files. Once the table is imported into the vehicle control software, it can be used to regulate the actions of the vehicles as they approach intersections, making sure that the vehicles do not enter an intersection at an unauthorized time.

- As it stands the image processing algorithm does not actually determine the absolute location of the tags in the image, only the location within the context of the image. It would be necessary to use static tags on the testbed to define a coordinate system in inches and translate a tag's location in the image into location in inches on the testbed given a defined origin. This would not be difficult to implement. The algorithm would only require a simple ratio calculation based on the distance between two static tags in pixels compared to the actual distance in inches. This ratio could then be applied to any distance value in the image to translate to inches.

- In addition, there is no current command created for the vehicles to make them stop on command, without stopping permanently. This is necessary for the vehicles to stop at an intersection where the "stoplight" is red, for example. As it stands, the vehicles can only move continuously. It would be necessary to add and implement a command called "brake" such that when it is called, the vehicle would come to a stop and wait for an appropriate input command to being moving again.

- The "traffic circle" command has not yet been developed. This command could be difficult to implement correctly, because the motion around the traffic circle is defined by the current location and the location of the exit to take. The motion around the circle will be predefined. The exact solution for how to implement this has not been developed yet.

- Finally, as it stands, the vehicle control software only supports control for one vehicle at a time. This is because normal execution of code follows a linear path, and control of multiple vehicles would need to be performed simultaneously. This can be achieved by implementing a process known as "multithreading." Multithreading essentially creates several new lines of execution called a "thread." Each thread executes a specific function, and once complete the thread is deleted and execution continues as normal. An algorithm that creates new threads corresponding to different car numbers has already been developed, but has not yet been implemented in the vehicle control software. This is because the variables that are used by the test execution portion of the vehicle control software would need to be translated to be global variables, and this would require some time to restructure the code to accommodate this. However, it is easily possible to do this and would not be too difficult to implement.

Once the initial testbed development is completed, future work could include adding features to improve the model, both for functional and aesthetic purposes. Improving the aesthetic appearance of the testbed would make the testbed more impressive for showcase purposes during open-house events. Some ways that the testbed could be improved aesthetically include:

- Using "grass" that is often used on model train platforms

- Using small model buildings similar to those used on model train platforms
- Adding “Stop” signs or “Yield” signs on the sides of the roads
- Using small orange cones or similar construction paraphernalia to block roadways
- Adding stoplights using LEDs and possibly a microcontroller for state control
- Running the USB cables for the cameras through the PVC pipes in the frame

These features would all improve the realism of the model.

Functionally speaking, one improvement that could be made to the testbed would be to create pre-defined road segment definitions. Each section or roadway that a vehicle can travel down only in one direction, i.e. without turning at an intersection, could be considered a road segment. Each road segment could be defined as a set of coordinates, and imported into the vehicle control software as a lookup table, similar to the aforementioned method for defining intersections and their states. This would allow vehicle commands to be expressed in terms of travelling along road segments, rather than travelling from point to point.

Some further improvements can be made to the testbed so that it can model more vehicle responses. For example, code could be added to the vehicle control software that would allow the vehicles to simulate collision avoidance sensors on the vehicles. This could be done by defining conditions for vehicle response if another vehicle is within a certain proximity from the vehicle under control. This would involve the image processing algorithm for vehicle location, and would certainly require high precision in location and fast software response times. Such algorithms would allow researchers to observe the behavior active safety system behavior for intelligent vehicles. Active safety systems include systems such as collision avoidance, radar

systems, anti-lock brakes and many more systems that operate to maintain vehicle safety on a real-time basis. Researchers are very interested in observing the behavior of these systems for the purpose of future development.

Conclusion

The focus of this project was to investigate whether a miniature scale testbed would provide a representative model for vehicle behavior. While the development of the testbed is still incomplete, no evidence exists to suggest that the successful implementation of the testbed is not possible in the miniature scale. Unfortunately, several issues need to be analyzed and dealt with before the testbed can become operational, but several possible solutions have been highlighted to deal with these issues. Once these issues are resolved, the MVT can be developed to allow researchers to gain useful insight into the operation of active safety systems in intelligent transportation vehicles.

Bibliography

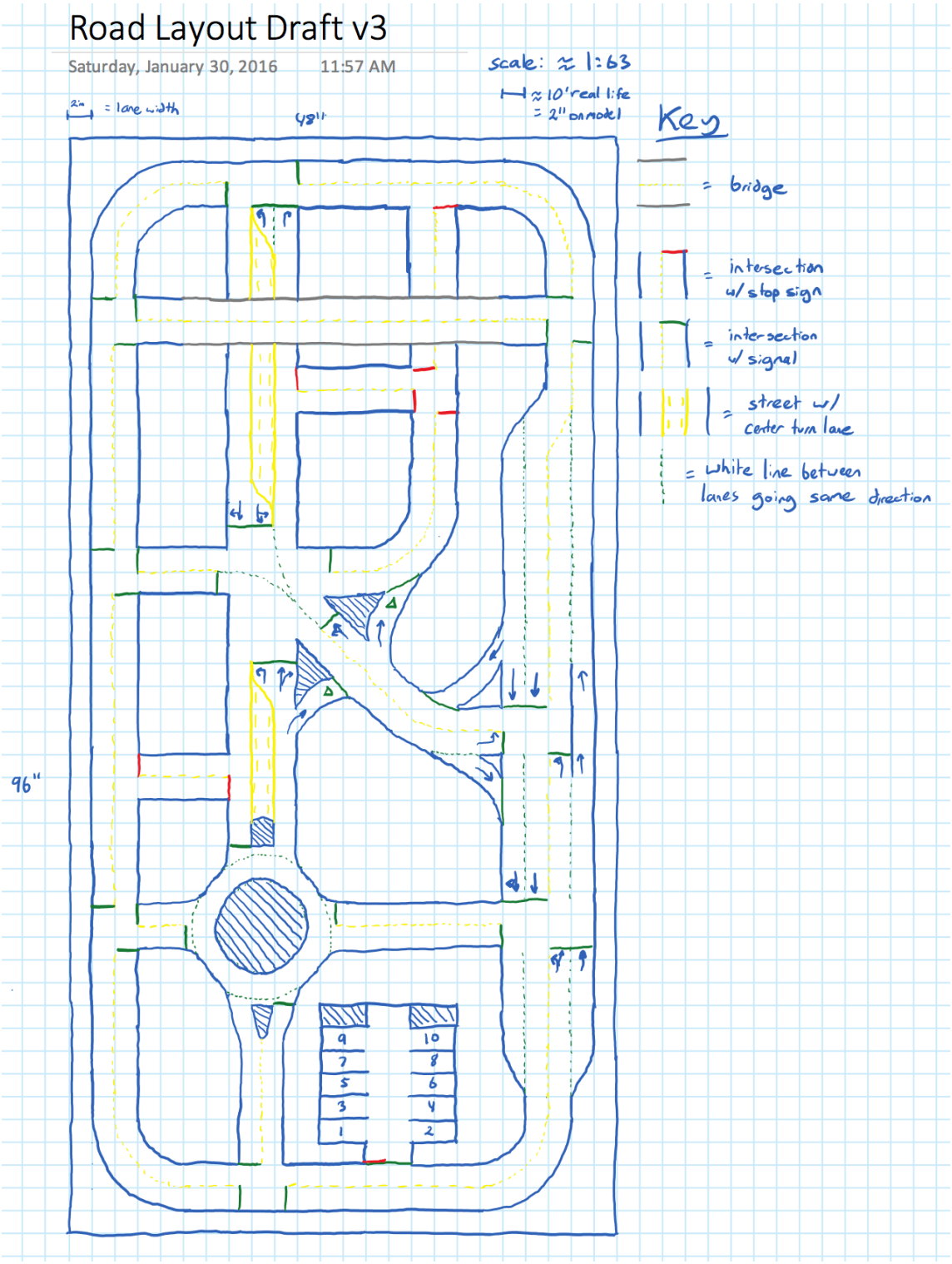
Berg, Alex. "Lecture 12 - Local Feature Detection." (n.d.): n. pag. *Massachusetts Institute of Technology*. Massachusetts Institute of Technology. Web. 27 Apr. 2016. <http://alumni.media.mit.edu/~maov/classes/comp_photo_vision08f/lect/18_feature_detectors.pdf>.

"Clustering - K-means." *A Tutorial on Clustering Algorithms*. N.p., n.d. Web. 27 Apr. 2016. <http://home.deib.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html>.

Derpanis, Konstantinos G. "Overview of the RANSAC Algorithm." *Overview of the RANSAC Algorithm* 1 (n.d.): n. pag. *Electrical Engineering and Computer Science*. Lassonde School of Engineering, 13 May 2010. Web. 27 Apr. 2016. <http://www.cse.yorku.ca/~kosta/CompVis_Notes/ransac.pdf>.

Appendices

Appendix A: The Testbed Roadway Design:



Appendix B: Matlab Code for the image processing algorithm:

findTags.m

```
%Miniature Vehicle Testbed for Intelligent Transportation Systems
%This code is to put together a final algorithm for the
%software to be used for the image processing portion of the project.
%In a nutshell, this code:
%   1. Imports an image
%   2. Converts it to grayscale
%   3. Detects corner objects using Harris Corner Detection
%   4. Uses k-means clustering to cluster the corners
%   5. Uses a RANSAC algorithm to detect lines from the corner clusters
%   6. Uses further RANSAC to detect the desired X pattern (tag detection)
%   7. Finds sets of three detected tags to find the symbols
%   8. Finds the identifier tag on the symbol and reads it
%   9. Collects and stores information from tag (location, orientation, ID)
%Writing began 11/19/15
%Written by Jared Suter
%Updated 4/9/16

clc
clear
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Parameters Used in the code
%numberClusters is the number of clusters to be created by k-means
numberClusters = 13;
%iterNum is the number of iterations to be performed in RANSAC
iterNum = 3;
%thDist is the maximum acceptable distance for a point to support a line
thDist = 6;
%supportTh is the threshold of points necessary to support a line
supportTh = 5;
%loopLimit is the maximum allowed loops (infinite loop prevention)
loopLimit = 25;
%sideRange is the range of values for a valid side of the symbol
sideRange = [230 245];
%hypRange is the range of values for a valid hyp of the symbol
hypRange = [330 340];
%slopeE is the error allowed for the slope side to be considered vertical
slopeE = 5;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Load Research Test Figure into workspace
testFigure = imread('IP Test Figure 16.jpg');
%Convert Picture to grayscale for processing
grayFigPre = rgb2gray(testFigure);
%Apply smoothing filter to the image
grayFig = imgaussfilt(grayFigPre,2);
%Use Harris Corner Detection to find corners
 [~, r, c] = harris(grayFig,1,900,1,1);%Output the number of corners detected
corners = [c,r];
numberCorners = length(corners);
fprintf(1,'Detected %d corners in IP Test Figure\n', numberCorners);
%Plot the corners
```

```
figure(1)
subplot(1,2,1);
scatter(corners(:,1), -1.*corners(:,2));
title('IP Test Figure Corners');
xlabel('X');
ylabel('Y');
%Show the corner detection on original image
subplot(1,2,2);
imshow(grayFig);
image(grayFig);
hold on;
scatter(corners(:,1), corners(:,2));
title('IP Test Figure w/ Corners');
xlabel('X');
ylabel('Y');
hold off;

%Create clusters using k-means clustering
[idx,C] = kmeans(corners, numberClusters);
%Show the clusters in a subplot on the image
figure(2);
subplot(1,2,1);
imshow(grayFig);
image(grayFig);
hold on
for i = 1 : numberClusters
    plot(corners(idx==i,1),corners(idx==i,2),'.','MarkerSize',25)
end
%Plot the centroids of the clusters
plot(C(:,1),C(:,2),'kx','MarkerSize',15,'LineWidth',3)
title('Cluster Assignments and Centroids');
xlabel('X');
ylabel('Y');
hold off

%Create plot for tagClusters
subplot(1,2,2);
imshow(grayFig);
image(grayFig);
title('Clusters with Tags');
xlabel('X');
ylabel('Y');
hold on

%Create array to hold indices of clusters with tags, intersections, and tag
counter
tagClusters = zeros(numberClusters,1);
tagIntersection = zeros(numberClusters,2);
numberTags = 0;

%RANSAC algorithm
for cluster = 1:numberClusters
    %Use the points from the first cluster - as example
    pts = [corners(idx==cluster,1) corners(idx==cluster,2)];
    %Find the number of points in the cluster
    ptNum = size(pts,1);
    %Initialize the size of the first line to 0
```

```
    line1Size = 0;
    %Initialize vectors for the indices of the valid lines and the support
    %values of the valid lines
    indices1 = zeros(iterNum, 2);
    support1 = zeros(iterNum, 1);
    %Initialize loop variables iter1 and p1
    %iter1 limits the iterations of the while loop to avoid infinite loops
    iter1 = 0;
    p1 = 0;
    if(ptNum > supportTh)
    while p1 < iterNum && iter1 < loopLimit %Loop until the specified number
of lines are found
        %Use a random index generator to get a random index of a corner
        sampleIdx = randsample(ptNum,2);
        %Get the random point from the cluster
        ptSample = pts(sampleIdx,:);
        %Reset the line1Size to zero
        line1Size = 0;
        %Formula for distance from a line: uses a1 and b1 below
        a1 = [ptSample(1,1)-ptSample(2,1) ptSample(1,2)-ptSample(2,2) 0];
        %Calculate b1 and distance for all the points in the cluster to find
        %those within the line support threshold
        for c1 = 1:ptNum
            %Calculate b1 from the distance formula
            b1 = [pts(c1,1)-ptSample(2,1) pts(c1,2)-ptSample(2,2) 0];
            %Calculate the distance based on a1 and b1
            dist1 = norm(cross(a1,b1)) / norm(a1);
            %If the distance is less than the threshold support distance
            if (dist1 < thDist)
                %Increment the support count for the line
                line1Size = line1Size + 1;
            end
        end
        %After the support for the line has been calculated
        %If the line1Size (support value) is higher than the support
threshold
        if (line1Size > supportTh)
            %Increment p1
            p1 = p1+1;
            %Save the values of the indicies for the lines and the support
            %value
            indices1(p1,1) = sampleIdx(1);
            indices1(p1,2) = sampleIdx(2);
            support1(p1) = line1Size;
        end
        %Increment the iteration value (to prevent infinite looping)
        iter1 = iter1+1;
    end

    %Find the support value and the index for the line with the most support
    [supportLine1, bestLine1] = max(support1);
    %If the supportLine1 value is greater than zero, a valid line was found
    %Now test to see if there is a second line to support the desired
tracking
    %symbol
    if supportLine1 > 0
        %Get the indices for the first line
        indicesLine1 = [indices1(bestLine1,1) indices1(bestLine1,2)];
```

```
        %Calculate the slope and the intercept for the first line y=Mx+B
        bestLine1M = (pts(indicesLine1(2),2) -
pts(indicesLine1(1),2))/(pts(indicesLine1(2),1) - pts(indicesLine1(1),1));
        bestLine1B = (-
1*bestLine1M*pts(indicesLine1(2),1))+pts(indicesLine1(2),2);
        %Calculate the ideal slope of the second line (M2 = -1/M1)
        idealLine2M = -1/bestLine1M;
        %Calculate a2 based on the desired slope
        %a2 = [sign(idealLine2M) bestLine1M 0];
        a2 = [1 idealLine2M 0];
        %Initialize loop variables iter2 and p2
        iter2 = 0;
        p2 = 0;
        %Initialize arrays to save the values of the valid indices and the
        %support values
        indices2 = zeros(iterNum, 1);
        support2 = zeros(iterNum, 1);
        %Initialize line2Size to 0
        line2Size = 0;
        while p2 < iterNum && iter2 < loopLimit %Loop until the specified
number of lines are found
            %Get a random index for the sample
            sampleIdx2 = randsample(ptNum,1);
            %Get the random point from the cluster
            ptSample2 = pts(sampleIdx2,:);
            %Reset line2Size
            line2Size = 0;
            %Iterate through the cluster to find support for the line created
by
            %the sample point
            for c2 = 1:ptNum
                %Calculate b2 from the distance formula
                b2 = [pts(c2,1)-ptSample2(1) pts(c2,2)-ptSample2(2) 0];
                %Calculate the distance using a2 and b2
                dist2 = norm(cross(a2,b2)) / norm(a2);
                %If dist2 is within the support distance threshold
                if (dist2 < thDist)
                    %Increment line2Size for the support value
                    line2Size = line2Size + 1;
                end
            end
            %After the line support is calculated, check if it is higher than
            %the support threshold
            if (line2Size > supportTh)
                %Increment p2 and save the index and support value
                p2 = p2+1;
                indices2(p2) = sampleIdx2;
                support2(p2) = line2Size;
            end
            %Increment iter2 to avoid the infinite loop
            iter2 = iter2+1;
        end
        %Find the saved line with the most support
        [supportLine2, bestLine2] = max(support2);
        %If suuportLine2 is greater than zero than it is a valid line
        if supportLine2 > 0
            %Obtain the index for the line anc calculate the intercept
            indexLine2 = indices2(bestLine2);
```

```
            bestLine2B = (-
1*idealLine2M*pts(indexLine2,1))+pts(indexLine2,2);
            %Plot the cluster

plot(corners(idx==cluster,1),corners(idx==cluster,2),'c.','MarkerSize',20)
            %Plot the lines
            fplot(@(x) bestLine1M*x+bestLine1B,[0,4000], 'b')
            fplot(@(x) idealLine2M*x+bestLine2B,[0,4000], 'r')
            %Calculate the intersection of the lines and plot the
intersection
            IntersectionX = (bestLine2B - bestLine1B)/(bestLine1M -
idealLine2M);
            IntersectionY = bestLine1M*IntersectionX + bestLine1B;
            plot(IntersectionX,
IntersectionY,'kx','MarkerSize',15,'LineWidth',3);
            %Tag is found! Increment the number of tags and add info to
            %arrays for tagClusters and tagIntersection
            numberTags = numberTags + 1;
            tagClusters(numberTags) = cluster;
            tagIntersection(numberTags,:) = [IntersectionX IntersectionY];
        end
    end
    end
end
%Initialize and create a table of distances from each center to another
centerDist = zeros(numberTags,numberTags);
for DC1 = 1:numberTags
    for DC2 = DC1:numberTags
        centerDist(DC1,DC2) = sqrt((tagIntersection(DC2,1)-
tagIntersection(DC1,1))^2 + (tagIntersection(DC2,2)-
tagIntersection(DC1,2))^2);
    end
end
%Count the number of symbols in the image based on tags
symbolCount = floor(numberTags/3);
symbolPoints = zeros(symbolCount,6);
%Create the symbols
a = 1;
symbolsFound = 0;
%Loop through the tags to find symbol groupings
while a < numberTags+1 && symbolsFound < symbolCount
    %Count the points valid to create a symbol
    points = 1;
    %Create variable for the indices of valid points
    point1 = 0;
    point2 = 0;
    %Create a variable to track whether the original point is the center or
    %not
    center = false;
    b = a;
    %Check the rest of the tags
    while b < numberTags+1 && points ~= 3
        c = centerDist(a,b);
        %If the center distance is within the valid range for a side
        if ((c > sideRange(1) && c < sideRange(2)) && point1 == 0)
            %Set the first valid point, and increment the point counter
            point1 = b;
            points = points+1;
```

```
            %Set the distances to 1 so they won't be reused
            for z = 1:numberTags
                centerDist(b,z) = 1;
                centerDist(z,b) = 1;
            end
        %If the center distance is within the valid range for the
        %hypotenuse
        elseif (c > hypRange(1) && c < hypRange(2))
            %Set the second valid point and increment the point counter
            point2 = b;
            points = points+1;
            %Set the distances to 1 so they won't be reused
            for z = 1:numberTags
                centerDist(b,z) = 1;
                centerDist(z,b) = 1;
            end
        %If the center distance is within the valid range for the side, and
        %the first side was already found
        elseif ((c > sideRange(1) && c < sideRange(2)) && point1 ~= 0)
            %Set the second valid point and increment the point counter
            point2 = b;
            points = points+1;
            %The original point is the center point, set center to true
            center = true;
            %Set the distances to 1 so they won't be reused
            for z = 1:numberTags
                centerDist(b,z) = 1;
                centerDist(z,b) = 1;
            end
        end
        %Increment to the next tag
        b = b+1;
    end
    %If 3 valid tags were found - valid symbol is found
    if points == 3
    %Set the points to the symbol matrix
        %If the original point was the center
        if (center)
            symbolPoints(a,1:2) = tagIntersection(point1,:);
            symbolPoints(a,3:4) = tagIntersection(a,:);
            symbolPoints(a,5:6) = tagIntersection(point2,:);
        else
            symbolPoints(a,1:2) = tagIntersection(a,:);
            symbolPoints(a,3:4) = tagIntersection(point1,:);
            symbolPoints(a,5:6) = tagIntersection(point2,:);
        end
        %Increment the counter for the number of symbols found
        symbolsFound = symbolsFound+1;
    end
    a = a+1;
end
%If symbols were found, create the symbol matrix
if symbolsFound > 0
    symbols = TestbedTag.createTagArray(symbolsFound);
    %For all the symbols found in the image
    for s = 1 : symbolsFound
        %Find the orientation of the symbol
        %Find the equation of the hypotenuse, and the angle of the hypotenuse
```

```
        hypM = (symbolPoints(s,6)-symbolPoints(s,2))/(symbolPoints(s,5)-
symbolPoints(s,1));
        hypTheta = atand(hypM);
        hypB = (-1*hypM*symbolPoints(s,5))+symbolPoints(s,6);
        %Find the midpoint and length of the hypotenuse
        hypMid = [(symbolPoints(s,1) + symbolPoints(s,5))/2
(symbolPoints(s,2) + symbolPoints(s,6))/2];
        hypLength = sqrt((symbolPoints(s,5)-symbolPoints(s,1))^2 +
(symbolPoints(s,6)-symbolPoints(s,2))^2);
        %If the center point is above the hypotenuse, then add 45 degrees to
        %find the orientation of the symbol
        if (symbolPoints(s,4) > hypM*symbolPoints(s,3) + hypB)
            symbolTheta = hypTheta + 45;
            orient = 1;
        else %If it is below, subtract 135 degrees
            symbolTheta = hypTheta - 135;
            orient = -1;
        end
        %Set the Orientation and Location of the TestbedTag object
        symbols(s).Orientation = symbolTheta;
        symbols(s).Location = hypMid;
        %Using the angle, find the slope of the symbol
        symbolM = tand(symbolTheta);
        %Find the slope of the first side of the symbol
        sideM = (symbolPoints(s,6)-symbolPoints(s,4))/(symbolPoints(s,5)-
symbolPoints(s,3));
        %If the slope is the same as the symbol orientation, that side is the
        %one on the right side of the original symbol
        if (abs(abs(sideM)-abs(symbolM)) < slopeE)
            %Set the vertical and horizontal lengths accordingly
            vLength = sqrt((symbolPoints(s,5)-symbolPoints(s,3))^2 +
(symbolPoints(s,6)-symbolPoints(s,4))^2);
            hLength = sqrt((symbolPoints(s,3)-symbolPoints(s,1))^2 +
(symbolPoints(s,4)-symbolPoints(s,2))^2);
        else
            vLength = sqrt((symbolPoints(s,3)-symbolPoints(s,1))^2 +
(symbolPoints(s,4)-symbolPoints(s,2))^2);
            hLength = sqrt((symbolPoints(s,5)-symbolPoints(s,3))^2 +
(symbolPoints(s,6)-symbolPoints(s,4))^2);
        end
        %Find the equation of the symbol, and plot the line
        symbolB = (-1*symbolM*hypMid(1))+hypMid(2);
        fplot(@(x) symbolM*x+symbolB,[0,4000], 'y');
        %Find the ID Tag
        %Bar distances horizontally are: 444, 398, 352, 306, 260
        %Find the scale of the detected image vs. the original image
        %In the original image, the length of the side is 400 pixels
        hScale = hLength/400;
        %Compute the locations of the bars (horizonally)
        hBarLoc = [hScale*444 hScale*398 hScale*352 hScale*306 hScale*260];
        %Initialize an array for the ID code
        idCode = zeros(1,5);
        %Parameter for the threshold for "1" vs "0"
        oneThresh = 150;
        %For all the bar locations, check if the bar is present
        for bar = 1:5
            %Compute the location of where the bar would be
            barLocX = round(symbolPoints(s,3)+sind(symbolTheta)*hBarLoc(bar)-
```

```
cosd(symbolTheta)*vLength);
            barLocY = round(symbolPoints(s,4)-cosd(symbolTheta)*hBarLoc(bar)-
sind(symbolTheta)*vLength);
            %If the intensity of the image is below the threshold, then the
bar
            %is present
            if grayFig(barLocY,barLocX) < oneThresh
                idCode(bar) = 1;
            end
            %Plot the marker to show where the bar should be
            plot(barLocX, barLocY,'kx','MarkerSize',15,'LineWidth',3);
        end
        %Calculate the decimal ID based on the binary code
        id = idCode(1)*16+idCode(2)*8+idCode(3)*4+idCode(4)*2+idCode(5);
        %Set the ID value in the symbol array
        symbols(s).ID = id;
    end
end
hold off
```

TestbedTag.m

```
classdef TestbedTag
    %This class is used to track symbols
    %   This class was created to help track symbols for the Miniature
    %   Vehicle Testbed for Intelligent Transportation Systems

   properties
       ID           %The ID of the symbol
       Location     %The location of the symbol in the image
       Orientation  %The orientation of the symbol in the image
   end
   methods
       function obj = TestbedTag
           %The constructor for this class.  Creates an array of S elements
           %of TestbedTag objects, intializing ID, Location, and
           %Orientation to 0, [0 0], and 0 respectively.
           obj.ID = 0;
           obj.Location = [0 0];
           obj.Orientation = 0;
       end
   end
   methods (Static)
       function obj = createTagArray(S)
          if nargin ~= 0
               obj = TestbedTag;
               obj(1,S) = TestbedTag;
               for c = 1:S
                   obj(c) = TestbedTag;
               end
           end
       end
   end
end
```

harris.m

```
% HARRIS - Harris corner detector
%
% Usage:  [corner] = harris1(im, sigma, thresh, radius)
%
% Arguments:
%            im     - image to be processed.
%            sigma  - standard deviation of smoothing Gaussian. Typical
%                     values to use might be 1-3.
%            thresh - threshold (optional). Try a value ~1000.
%            radius - radius of region considered in non-maximal
%                     suppression (optional). Typical values to use might
%                     be 1-3.
%
% Returns:
%            corner - array of corner point coordinates
%
% If thresh and radius are omitted from the argument list 'cim' is returned
% as a raw corner strength image and r and c are returned empty.

% Reference:
% C.G. Harris and M.J. Stephens. "A combined corner and edge detector",
% Proceedings Fourth Alvey Vision Conference, Manchester.
% pp 147-151, 1988.
%
% Author:
% Peter Kovesi
% Department of Computer Science & Software Engineering
% The University of Western Australia
% pk@cs.uwa.edu.au  www.cs.uwa.edu.au/~pk
%
% March 2002
%
% Adapted for use with the Miniature Vehicle Testbed for Intelligent
% Transportation Systems by:
% Jared Suter
% Department of Electrical and Computer Engineering
% The Ohio State University
%
% April 2016

function [corners] = harris(im, sigma, thresh, radius)

narginchk(4,4);

dx = [-1 0 1; -1 0 1; -1 0 1]; % Derivative masks
dy = dx'; %'

Ix = conv2(im, dx, 'same');    % Image derivatives
Iy = conv2(im, dy, 'same');

% Generate Gaussian filter of size 6*sigma (+/- 3sigma) and of
% minimum size 1x1.
g = fspecial('gaussian',max(1,fix(6*sigma)), sigma);
```

```
Ix2 = conv2(Ix.^2, g, 'same'); % Smoothed squared image derivatives
Iy2 = conv2(Iy.^2, g, 'same');
Ixy = conv2(Ix.*Iy, g, 'same');

cim = (Ix2.*Iy2 - Ixy.^2)./(Ix2 + Iy2 + eps); % Harris corner measure

% Alternate Harris corner measure used by some.  Suggested that
% k=0.04 - I find this a bit arbitrary and unsatisfactory.
%   cim = (Ix2.*Iy2 - Ixy.^2) - k*(Ix2 + Iy2).^2;

% Extract local maxima by performing a grey scale morphological
% dilation and then finding points in the corner strength image that
% match the dilated image and are also greater than the threshold.
sze = 2*radius+1;                   % Size of mask.
mx = ordfilt2(cim,sze^2,ones(sze)); % Grey-scale dilate.
cim = (cim==mx)&(cim>thresh);       % Find maxima.

[r,c] = find(cim);                  % Find row,col coords.

%Remove corners near the edge of the image
maxX = size(im,1);
maxY = size(im,2);
inset = 20;
numCorn = 0;
inCorn = 0;
for q = 1:size(r,1)
   if ((c(q) > inset && c(q) < maxX-inset) && (r(q) > inset && r(q) < maxY-
inset))
       inCorn = inCorn + 1;
   end
end
corners = zeros(inCorn,2);
for q = 1:size(r,1)
   if ((c(q) > inset && c(q) < maxX-inset) && (r(q) > inset && r(q) < maxY-
inset))
       numCorn = numCorn + 1;
       corners(numCorn,:) = [c(q) r(q)];
   end
end

end
```

Appendix C: Commented sample test file

```
This is a sample to show the structure of the text input file that controls
the Miniature Vehicle Testbed for Intelligent Transportation Systems
Written by Jared Suter
Updated 4/13/16
Sample Program: "program1.txt"
4                               :Number of cars participating in the program
1 27,5                  :Car 1, start from 27,5
2 33,5                  :Car 2, start from 33,5
3 11,3                  :Car 3, start from 11,3
4 16,12                 :Car 4, start from 16,9
A 1                             :Intersection A is initially in state 1
0                               :Begin Program Commands
1 d 21,5                :Car 1, drive to 21,5
1 s 13,5                :Car 1, navigate straight to 13,5
1 d 10,5                :Car 1, drive to 10,5
1 c 9,5.25              :Car 1, curve to 9,5.25
1 c 8,___               :Car 1, curve to 8,____
1 c 7,___               :Car 1, curve to 7,____
1 c 6,___               :Car 1, curve to 6,____
1 c 5,10                :Car 1, curve to 5,10
1 d 5,23                :Car 1, drive to 5,23
1 p                             :Car 1, park (stop)
2 d 21,5                :Car 2, drive to 21,5
2 s 13,5                :Car 2, navigate straight to 13,5
2 d 10,5                :Car 2, drive to 10,5
2 c 9,5.25              :Car 2, curve to 9,5.25
2 c 8,___               :Car 2, curve to 8,____
2 c 7,___               :Car 2, curve to 7,____
2 c 6,___               :Car 2, curve to 6,____
2 c 5,10                :Car 2, curve to 5,10
2 d 5,23                :Car 2, drive to 5,23
2 p                             :Car 2, park (stop)
3 d 13,3                :Car 3, drive to 13,3
3 l 18,7                :Car 3, navigate left to 18,76
3 d 18,16               :Car 3, drive to 18,16
3 d 19,19               :Car 3, drive to 19,19
3 w 5                   :Car 3, wait for 5 seconds
3 t 22,25.75    :Car 3, take the traffic circle to the point 22,25.75
3 d 36,26               :Car 3, drive to 36,26
3 p                             :Car 3, park (stop)
4 d 16,8                :Car 4, drive to 16,8
4 r 13,5                :Car 4, navigate right to 13,5
4 d 10,5                :Car 4, drive to 10,5
4 c 9,5.25              :Car 4, curve to 9,5.25
4 c 8,___               :Car 4, curve to 8,____
4 c 7,___               :Car 4, curve to 7,____
4 c 6,___               :Car 4, curve to 6,____
4 c 5,10                :Car 4, curve to 5,10
4 d 5,23                :Car 4, drive to 5,23
4 p                             :Car 4, park (stop)
-1                              :end of program
```

Appendix D: The result of the image processing algorithm in Matlab using a test image:

Input image: IP Test Figure 16.jpg

Figure 17: The input test image for the Matlab image processing algorithm

Matlab Console Output:

```
Warning: NARGCHK will be removed in a future release. Use
NARGINCHK or NARGOUTCHK instead.
> In harris (line 40)
  In ImageProcessingStep3 (line 57)
Warning: CONV2 on values of class UINT8 is obsolete.
         Use CONV2(DOUBLE(A),DOUBLE(B)) or
CONV2(SINGLE(A),SINGLE(B)) instead.
> In uint8/conv2 (line 10)
  In harris (line 45)
  In ImageProcessingStep3 (line 57)
Warning: CONV2 on values of class UINT8 is obsolete.
         Use CONV2(DOUBLE(A),DOUBLE(B)) or
         CONV2(SINGLE(A),SINGLE(B)) instead.
> In uint8/conv2 (line 10)
  In harris (line 46)
  In ImageProcessingStep3 (line 57)
Detected 138 corners in IP Test Figure
```

Note: The warnings in the console output above do not actually have any effect on the algorithm. To remove the warnings would require some investigation on how to replace those functions.

Output Figure 1:

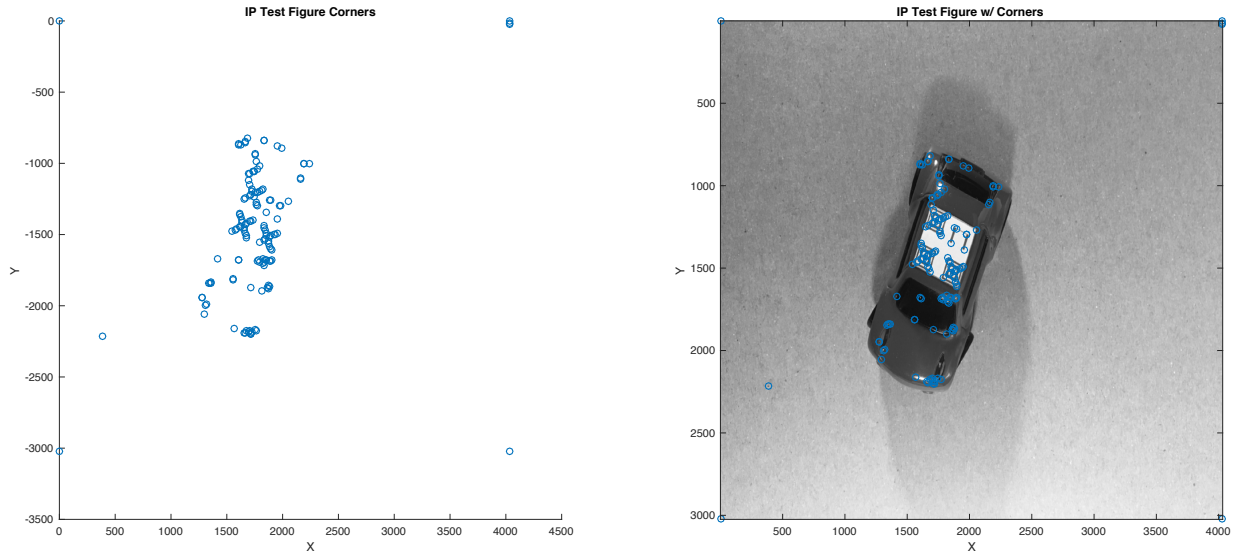


Figure 18: Matlab output Figure 1 from findTags.m

Output Figure 2:

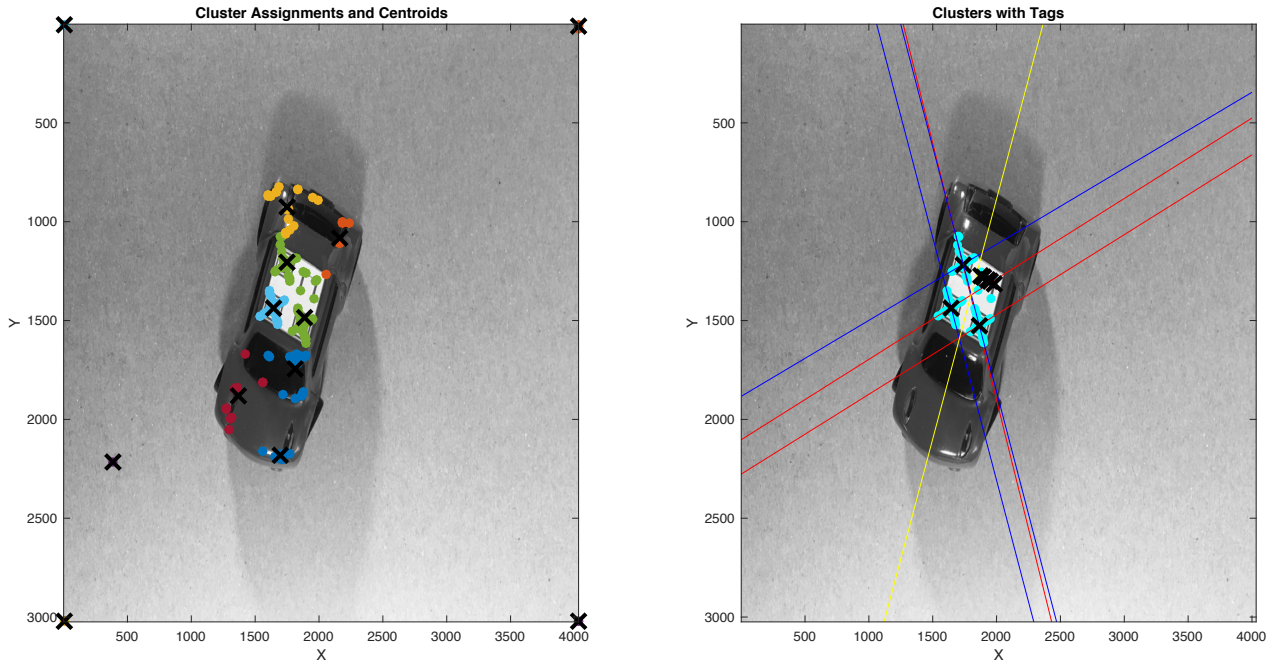


Figure 19: Matlab output Figure 2 from findTags.m

```
symbols =

  TestbedTag with properties:

            ID: 17
      Location: [1.8002e+03 1.3702e+03]
   Orientation: 112.3782
```

Figure 20: TestbedTag output to console

Output Figure 1 shows the detected corners in the image, and the corners overlaid on the original image in grayscale. Output Figure 2 shows the clusters and their respective centroids on the left, and the detected nested-square symbols, their lines in red and blue, the yellow line for orientation and the calculated location of the identification tag which is shown by the 'X's. Figure 20 shows the TestbedTag output to the console. This shows the decimal ID value 17 was correctly calculated from the tag, and the location in pixels and orientation in degrees of the tag in the image. Note that the y-axis in Figure 19 is inverted so that the angle is indeed 112.3782 degrees.